Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 193 (2007) 61–79

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Strength Induction in a Haskell Program Verifier

Richard B. Kieburtz

Portland State University
Portland, Oregon, USA

Abstract

Haskell employs a melange of strict and non-strict evaluation semantics, hence a Haskell verifier should be capable of checking assumptions that program variables may or may not denote well-defined values. The paper introduces a new strategy, called strength induction, that supports automatic checking of definedness assumptions.
Strength induction has been implemented in Plover, an automated property-verifier for Haskell programs that has been under development for the past three years as a component of the Programatica project. In Programatica, predicate definitions and property assertions written in *P-logic*, a programming logic for Haskell, can be embedded in the text of a Haskell program module. Properties refine the type system of Haskell but cannot be verified by type-checking alone; a more powerful logical verifier is required.
Plover codes the proof rules of P-logic, and additionally, embeds strategies and decision procedures for their application and discharge. It integrates a reduction system that implements a rewriting semantics for Haskell terms with a congruence-closure algorithm that supports reasoning with equality.

Keywords: Haskell, automatic verification, programming logic, rewriting, strategies, Stratego, reduction, normal forms, theorem proving, structure splitting, strength induction, lazy evaluation, strictness

1 Introduction

This paper describes a few of the strategies used in Plover, an automatic property verification tool for Haskell programs developed in connection with the Programatica project. Its objective is to demonstrate the feasibility of automatic verification of formally specified properties of computer programs.

Plover can provide assurance of many useful properties based upon the soundness of automated reasoning in a formal logic. Plover specifically implements reasoning in *P*-logic, which is the verification logic of Haskell98,

doi:10.1016/j.entcs.2007.10.008

whereas few other available proof assistants directly support a verification logic so closely tied to a wide-spectrum programming language. Although this paper is reasonably self-contained, a reader not familiar with Plover or with programmed strategies for verification may find it helpful to consult an earlier paper that provides a more comprehensive view [5].

1.1 Proof search by term rewriting

To verify a logical property, P, of a program fragment, e, in a logical context, Γ, a verifier can search for a formal proof of the hypothetical judgment $\Gamma \vdash e ::: P$, using the logical theory of the programming language. Here, the assertion has been expressed as a *sequent*, relating a logical context containing *assumptions* about properties of program variables bound the textual context of a program fragment to a proposition, $e ::: P$ that the fragment has a specified property.

A sequent that is to be verified can be written as a term in a suitably defined abstract syntax. The logical theory of the programming language can be expressed in a finite set of term-rewriting rules. When a rule is applicable to a term of interest, it can fire, transforming the term to one or more terms that represent antecedent assertions in the logical theory. These terms replace the original in the set of assertions that must be discharged to realize a verification proof. Whenever a term is produced that is an axiom of the theory, the assertion made by that term is discharged without further rewriting. A verification of the original assertion is a sequence of rewrites that ultimately discharges all terms of interest.

However, term-rewriting is not a linear process. Typically, several alternative rules that may fire to rewrite a given term of interest. In a "pure" term-rewriting system, selection among alternative rules is non-deterministic. For a computational implementation, even one that employs bounded parallelism, it is necessary to sequentialize the selection of alternatives. Backtracking when no further progress can be made along a previously selected path assures completeness of a search for a successful verification path. However, the performance of a verification system may depend crucially on the rule selections that it makes.

Herein lies the role of programmed strategies. While interactive theorem proving assistants can call upon a human user to select a promising search path, an automated verifier must rely upon pre-programmed strategies for guidance.

1.2 Strategies direct rewriting

A degree of control can be provided in an otherwise undirected term-rewriting system by incorporating side conditions for a rule to fire, to augment control by pattern-matching at a redex. However, the side conditions in a conditional rewrite can only refer to variables that are bound in matching the pattern of the rule to a redex. They cannot take into account contextual information or the computational "state" in which a possible rule application might occur. Thus it is difficult to specify policies such as top-down or bottom-up traversals of a term by conditional rewriting alone.

A further generalization of control in a rewriting system can be provided by explicitly programmed strategies. Strategies may incorporate conditional rewriting rules but can also specify sequencing of rewriting steps and can prescribe alternative steps to be tried if a rewriting step fails [1]. Furthermore, programmed strategies can be recursively defined and can be parameterized over other strategies. Strategy parameters can carry information from the context surrounding a potential redex, to condition the application of a rewrite rule.

1.3 Roadmap

Section 2 introduces some basic notations used in $P - logic$, the verification logic for Haskell. Sections 3 and 4 provide a brief introduction to the architecture of *Plover*—the automated prototype Haskell verifier, and its implementation in Stratego. Section 5, which is taken from [5], is an extended example of programmed strategies. It presents several strategies for recognizing and calculating normal forms of an untyped lambda calculus. Sections 6 and 7 discusses a specific strategy, *structure splitting*, that is useful in reasoning about expressions typed in Haskell data types. A set of type-independent strategies that are useful in simplifying `let` expressions is given in Section 7. These strategies support the strategy of *strength induction* that is discussed in Section 8. Use of this strategy is illustrated by following the steps of an automated proof of one of the monad laws for a Haskell specification of a monad of state. Conclusions and a brief discussion of related work are given in Section 9.

[1] Strategies that specify sequencing and alternatives have long been used in interactive theorem-provers [8,10,7] to reduce the workload of a human user.

2 A verification logic for Haskell

P-logic [4] is a programming logic specific to Haskell, meaning that the object-language terms whose properties can be asserted are expressions in Haskell98, well-typed in the context of the program module in which assertions appear. Its proof rules are consistent with a denotational semantics for Haskell98, enabling so-called total correctness assertions about any legal Haskell98 program to be formulated in *P*-logic.

Other examples of language-specific verification logics are ACL2 [3], a verification logic for Common Lisp, and Sparkle [2], a verifier for Clean 2.0. When assertions are formulated in a language-specific verification logic it is unnecessary to translate expressions and their asserted properties into another logical formalism, which may have a different type system, and with the attendant risk that errors may be introduced in the translation.

2.1 Predicates refine types

Every predicate form definable in *P*-logic is subject to a typing discipline: a predicate is the refinement of a Haskell type. *P*-logic provides basic constructions for unary predicates analogous to the constructors of Haskell types. Predicate constructions are formed with the arrow constructor (`->`), finite tupling, predicate constructor application and predicate disjunction, which is analogous to the sum-of-constructions by which data types are defined. Additional predicate constructions go beyond the constructions of Haskell types. These include predicate disjunction [2], predicate negation [3], predicate abstraction, least and greatest fixed-point constructions, and comprehensions that utilize formulas with quantified object variables in the specification of a predicate.

Data constructors are implicitly lifted to predicate constructors. A data constructor typed as $C :: \tau_1 \to \cdots \to \tau_k \to \tau$ becomes a predicate constructor typed as $C :: (\tau_1 \to \mathsf{Prop}) \to \cdots \to (\tau_k \to \mathsf{Prop}) \to \tau \to \mathsf{Prop}$. A predicate, $C\, P_1 \cdots P_k$ is satisfied by a Haskell expression that semantically reduces to a head normal form $C\, e_1 \cdots e_k$, provided that each of the argument expressions e_i satisfies its respective predicate, P_i.

There is a distinguished, polymorphically typed, binary predicate $(\,\mathsf{===}\,) :: \alpha \to \alpha \to \mathsf{Prop}$ that is interpreted as semantic equality of expressions. For more detail on the forms and meanings of predicates in *P*-logic, the reader is

[2] Predicate disjunctions are analogous to intersection types, which are not part of Haskell's type system.

[3] Predicate negation has no direct analogy as a type constructor, but is given a meaning in *P-logic* [4] that is compatible with the domain structure of a type frame.

referred to [4].

2.2 Well-definedness and the strength modality

Recall that in Haskell semantics, every type frame is a pointed cpo and furthermore, arrow, product and sum (data) [4] types are "lifted" above an undefined element that might not be expected in their categorical counterparts.

In consequence of this semantic structure, every unary predicate in P-logic is satisfied by the bottom (*undefined*) element in the type frame of its argument. Consequently, an assertion that an expression has an unannotated property, P, corresponds to a so-called *partial-correctness* assertion of the property that is intuitively understood by P. To express a total-correctness assertion, the symbol (\$) can be prefixed to a unary predicate expression. A predicate expression $\$P$ is satisfied only by an expression whose evaluation terminates in a form satisfying the property P. We call this the *strong modality* of P-logic.

3 The architecture of a verifier

In this section, we give an overview of the architecture of Plover—a custom verifier for P-logic. Plover operates as a verification server for Programatica. It relies upon the Programatica front-end tool, `pfe`, to parse, type-check and analyze dependencies of a Haskell module, generating input for Plover in the form of a list of terms in an abstract syntax for Haskell. `pfe` is run in a directory containing a number of Haskell source-code modules. It also has access to one or more Haskell libraries containing definitions that the source code may refer to.

3.1 Assertions as sequents

An assertion in P-logic is a propositional form in the context of a Haskell module. Sequents afford a representation of assertions that is particularly convenient for computational manipulation. The form of a Plover sequent is:

$$\mathcal{E}, \Gamma \vdash \Delta$$

where $\mathcal{E}$ is a finite set of type bindings (the *type environment*) for object variables that are in scope; Γ is a finite set of (implicitly conjoined) propositional

[4] Strictly speaking, a data type is a lifted sum only when its data constructors are not annotated in its declaration. If a data constructor is given a strictness annotation at some argument type, then the bottom element of that type frame is "coalesced" with the bottom element of the data type.

forms, called the *assumptions* of the sequent and Δ is a finite set of (implicitly disjoined) propositional forms, called the *conclusions* of the sequent.

A sequent is represented as a term in an abstract syntax. The entailment symbol ($\vdash$) is a top-level constructor of terms of sort Sequent. Finite sets are represented as lists but the order of listing is of no logical consequence. To economize on the exposition in this paper, we shall ignore the type bindings in a sequent.

The general form of a Programatica assertion is a proposition in prenex normal form. Typings of the free and quantifier-bound variables have been calculated by `pfe`. To translate an assertion into sequent form, typings of these variables are collected in a type environment after skolemizing existentially quantified variables. If the quantifier-free matrix of an assertion is an implication, the set of implicands constitutes the list of assumptions in its sequent representation. Finally, if the implicant is a disjunction of propositions, the disjuncts constitute the set of possible conclusions; otherwise the entire implicant is listed as a singleton set of conclusions.

3.2 Inference rules

An inference rule of *P*-logic relates a consequent assertion to a finite set of antecedent assertions, called the *verification conditions* for the consequent. A rule is sound if logical validity of the antecedents entails validity of the consequent. A rule with an empty set of antecedents is an axiom.

Inference rules can be coded as term-rewriting rules. Given that a sequent is coded in abstract syntax terms, an inference rule can be implemented as a rewrite from its consequent to a list of antecedents.

3.3 Terms of several sorts

Rewrite rules may be triggered by matching on redexes that involve terms of several sorts (see Fig. 1). Rewrites that depend only on terms of sort Sequent correspond to structural rules of sequent calculus. Rewrites that depend on terms of sorts Prop or Pred implement rules of propositional and predicate calculus that underlie *P*-logic. Rewrites that depend also on terms of sorts HTerm and HPattern implement rules that interpret Haskell semantics. A few rules analyze terms of sort HType to distinguish instances of Haskell type classes.

Sequent	—	terms representing sequents
Prop	—	propositional terms
Pred	—	predicate terms
HTerm	—	Haskell expressions
HPattern	—	Haskell pattern forms
HType	—	Haskell type expressions

Fig. 1. Sorts of terms used in Plover

4 The Plover verification engine

4.1 Stratego: a strategy implementation language

Plover is implemented in a strategy programming language—Stratego [13]. In Stratego, the notion of strategy is formalized as a first-class computational entity, just as are functions or procedures in most programming languages. The subject to which a strategy is applied is a term; the environment for a strategy application is a set of bindings of variables that may occur in terms.

A strategy application may succeed or fail. If it succeeds, it produces a new term and may extend the set of extant bindings. If it fails, the extant term and bindings are unchanged.

The elements of strategy composition are summarized in Fig. 2. In Stratego, strategy definitions can be abstracted on strategy parameters and can be recursive.

Stratego is a compiled language whose runtime system is built on top of the A-term library [11]. This system provides maximal sharing of terms, implemented by internal hashing. In effect, it implements a content-addressable store, indexed by terms themselves.

4.2 Proof discovery by term rewriting

Plover implements proof rules of *P*-logic as rewrites, transforming a goal term of sort Sequent into one or more sequents representing its verification conditions. This process continues until every verification condition can be discharged, either by recognizing it to be an instance of an axiom of *P*-logic or by recognizing its conclusion to be a reflexive equality or an instance of one of its assumptions.

- A *pattern* is a strategy that binds its variables to components of a matching term, extending the current environment.
- A *term-builder* is a form that constructs a new term, using variable bindings from the current environment.
- A *rewrite rule* is a strategy consisting of a pattern followed by a term-builder.
- A *conditional* strategy is a rewrite rule scoped over an auxiliary strategy. The rewrite succeeds only if its auxiliary strategy succeeds in the current environment as extended by pattern-matching of the rule.
- A *sequential* composition (`;`) executes two strategies in sequence. It succeeds only if both components succeed.
- A *choice* composition of strategies succeeds if either one of its components succeeds. A choice can be nondeterministic (`+`) or left-biased (`<+`).

Fig. 2. Elements of a strategy

4.3 Strategies + Decision procedures = Verifier

While the fundamental strategy used in an automatic verifier is term rewriting, computational performance can be improved by augmenting rewriting with cooperating decision procedures for some decidable sub-theories of the programming logic. Some decision procedures incorporated in Plover are described in a previous paper [5]. Plover's decision procedure implementation is derived from the Nelson-Oppen method for combining decision procedures [6].

5 Example: Strategies for normalizing terms

Plover is capable of reducing Haskell expressions to normal forms by applying reduction rules compatible with Haskell's denotational semantics. Strategies for normalization are so important to the success of a verifier that we shall spend some time discussing them. However, Haskell expressions have so many possible forms, including **let**, **case**, **if-then-else** expressions, records and data constructions, in addition to abstractions and applications, that we shall illustrate normalization strategies with a much simpler language—untyped lambda calculus with only the β-rule for reduction. Fig. 3 gives constructors for an abstract syntax of this language, in Stratego notation.

```
sorts Exp
constructors
   Var  :  String -> Exp
   Abs  :  String * Exp -> Exp
   App  :  Exp * Exp -> Exp
   Let  :  [(String * Exp)] * Exp -> Exp
```

(The constructor `Let` is used only to represent explicit substitutions.)

Fig. 3. Abstract syntax constructors for lambda calculus

```
strategies
   whnf = Abs(id,id) + rec r(Var(id) + App(r,id))
   hnf = rec s(Abs(id,id) + rec r(Var(id) + App(r,s)))
   snf = rec s(Abs(id,s) + rec r(Var(id) + App(r,s)))
```

Fig. 4. Recognition strategies for normal forms

5.1 Recognizing normal forms

The most common definition of a normal form for the lambda calculus is that reached by exhaustive application of its reduction rules (if the process terminates) at every possible redex. That's not the only useful definition, however. If reduction is suspended under abstractions, then the form reached is called a *head* normal form. If it is, in addition, suspended in the *rand* term of an application, exhaustive reduction stops at a *weak* head normal form. Let's see how each of these can be characterized with a simple recognition strategy.

A recognition strategy is like a pattern, but since it does not need to produce bindings of the pattern variables, it may be defined recursively, to match terms nested to arbitrary depth. Three such definitions are given in Fig. 4, each corresponding to one of the three specifications of normal forms for the lambda calculus that were mentioned in the preceding paragraph. The definitions are subject to the assumption that all terms are well-sorted, i.e. that a constructor is only applied to subterms of the sorts given in the constructor's signature.

Let's examine the first definition in detail. It consists of two alternative strategy components, the second of which has a recursive definition scoping

```
strategies
  Beta     =         \ App(Abs(Var(x),m),n) -> Let([(x,n')],m)
                       where <RenameBoundVarsIn> n => n'                  \

  LetElim  = rec r({ \ Let([],m) -> m                                     \
                   + \ Let([elmt | bindings],m) -> <r> Let(bindings,m')
                       where <Replace> (elmt,m) => m'                     \
                  })

  Replace  = rec r({ \ ((x,n),Var(x)) -> n                                \
                   + \ (elmt,App(m,n)) -> App(<r>(elmt,m),<r>(elmt,n))    \
                   + \ ((x,_),Abs(y,n)) -> Abs(y,<r>(elmt,n))
                       where <not(eq)> (x,y)                              \
                  <+ \ (_,e) -> e                                         \
                  })
```

Fig. 5. Reduction strategies with explicit substitution

over two alternatives. Notice the use of the data constructors as strategy constructors. A data constructor, when lifted to become a strategy constructor, is satisfied by a term built with the same data constructor and whose argument terms satisfy the respective strategy arguments given to the strategy constructor. Note also the use of the `id` strategy as an argument to a lifted strategy constructor. `id` is a library strategy that always succeeds, leaving the current term and bindings unchanged. It is analogous to a wildcard designator in a pattern.

The first alternative of the `whnf` strategy is satisfied by an `Abs` construction with any well-sorted subterms as arguments. The recursively defined alternative is satisfied by any `Var` term and also by an `App` term whose rator is either a `Var` term or an `App` term in weak head normal form. Thus the recognition strategy excludes any `App` term that has an `Abs` term as rator. There is no restriction on the rand subterm of an `App` construction.

The second definition is similar to the first, but adds the restriction that the rand of an `App` term must be in head normal form. Since the allowed forms of the rator and rand subterms differ, an additional level of recursive definition is needed to accommodate both forms. Finally, the third definition adds the restriction that the body of an `Abs` term must have the specified normal form, as well.

5.2 Reduction strategies

Reduction rules are readily programmed as alternative, conditional rewrites. Fig. 5 gives a Stratego encoding of rules for β-reduction with explicit, capture-

avoiding substitution. In Stratego notation, a rewrite rule is bracketed between backslash symbols.

In the syntax of a rule, a pattern appears to the left of the rewrite symbol (`->`) and to its right is a term-building strategy. If the rule is conditional, the condition strategy follows the keyword `where`. When a conditioning strategy succeeds it returns a result term. If it should fail, the attempted conditional rewrite fails. (A "pure" conditioning strategy occurs in the third alternative rule of the definition `Replace`. It is not followed by a pattern extension and is executed only to determine success or failure.)

When a strategy is enclosed in curly brackets, recursive invocations of the strategy bind fresh variables in auxiliary rule definitions, such as `m'` and `n'` in the definitions of `Beta` and `LetElim` above. Otherwise, bound variables would remain in scope throughout recursive invocations. Since a variable, once bound in a definition, cannot be rebound to a different term, failure to use curly brackets often results in failure of a recursively-defined strategy.

The strategy `RenameBoundVarsIn`, whose definition is not shown here, replaces every occurrence of the variable name bound in an `Abs` term with a fresh name. `RenameBoundVarsIn` uses a package of generic renaming strategies supplied in the Stratego library, specializing them to terms of sort `Exp`.

The pattern `[`*hd* `|` *tl*`]` matches a non-nil list, binding the pattern variable *hd* to the head of the list and *tl* to its tail. The library strategy `eq` is satisfied by a pair of syntactically identical terms.

Notice that the last alternative listed in the definition of `Replace` is separated by the operator symbol (`<+`) rather than the symbol (`+`). The symbol (`<+`) designates left-biased choice rather than nondeterministic choice of alternative strategies. Since the pattern of the final alternative would match an arbitrary pair of terms, it overlaps the patterns of each of the rules that precede it, and is programmed to fire only as a default alternative.

5.3 Normalization strategies

Finally, we are ready to present strategies to normalize lambda expressions, using the β-rule and explicit substitution. Three normalization strategies, corresponding to the three normal forms presented earlier, are shown in Fig. 6. Notice that these are defined with data constructors implicitly lifted to strategy constructors, as were the recognition strategies, rather than with rewrite rules, as were the reduction strategies.

Each normalization strategy first tries the recognition strategy for its respective normal form, returning immediately in case the current term is normalized. Otherwise, if the current term matches an `App` construction, then

```
strategies
  BetaSubst    =  Beta; LetElim
  Lazy-eval    =  rec r(whnf<+ App(r,id); try(BetaSubst; r))
  Eager-eval   =  rec r(hnf <+ App(r,r); try(BetaSubst; r))
  Strong-eval  =  rec r(snf <+ Abs(id,r)
                               + App(r,r); try(BetaSubst; r))
```

in which `try`(1) is a library strategy defined as

```
try(s) = s <+ id
```

which executes `s` if it succeeds and otherwise executes the identity strategy.

Fig. 6. Three strategies for normalization

`BetaSubst` is tried after normalizing appropriate subterms of the construction. In case `BetaSubst` succeeds, it is still not assured that the result term is normalized, thus the normalization strategy is applied recursively to the result.

Plover incorporates strategies similar to `Lazy-eval` and `Eager-eval` but for Haskell terms. Both lazy and strict normalization strategies are used repeatedly by Plover and interact with a partial decision procedure for semantic equality to simplify Haskell terms.

6 Strategies specific to structure-determining types

Many of the strategies used in Plover are designed to simplify expressions or assertions that are specific to a Haskell type, or type constructor. This section explains two such strategies and suggests how they might be generalized.

6.1 Structure splitting

Some types uniquely determine the top-level structure of normal-form terms of the type. Product types (finite tuples) and data types with only a single constructor have this property. Structure-determining types see greater use in Haskell programs than would be the case in many other languages.

When the argument of an application, the definiens of a local definition or the scrutinee of a case expression can be assumed to be equal to a head normal form of the type, subject to the auxiliary verification condition that the given expression of the type is well defined. We say that an expression

has a *structure-determining type* if the constructor of a head normal form of the type is unique. When an expression has a structure-determining type, the assumed equality of the expression to a head normal form can be specified as an explicit, new assumption. To synthesize a term in head normal form, the unique data constructor is applied to a fresh variable in each of its argument positions.

This strategy, which we call *structure splitting*, is used by Plover whenever the strategy might enable further strategies of application reduction, **case** reduction, or **let** reduction (inlining of local definitions).

6.2 Eliminating equalities of abstractions

When the conclusion of a sequent asserts an equality between an explicit abstraction and another expression (necessarily of the same type), the abstraction can be eliminated by applying both sides of the asserted equality to a common, fresh variable. If the abstraction pattern happens to be of a structure-determining type, structure splitting may enable immediate reduction of the synthetic application.

7 Generic strategies for non-recursive let expressions

Several strategies employed in Plover cannot be called type-specific, yet they deal with program constructions that are particular to Haskell. These include strategies for local definitions introduced in **let** or **where** clauses, guarded expressions, fixed-point induction and strategies that instantiate quantified assumptions. We shall describe here the strategies Plover uses for **let** expressions and to resolve sequents that depend upon well-definedness of specific expressions.

A **let** expression constitutes a list of local definitions that scope over a single object expression. The order in which definitions are listed is semantically unimportant in Haskell, as they scope over one another, as well. Thus a set of definitions may be mutually recursive.

A **let** expression can be simplified by manifesting the equalities entailed by its definitions, whenever possible. Several strategies for **let** expressions and local definitions, implemented in Plover, are described informally in Figure 7. The formal specification of these strategies in Stratego is fairly extensive, and embodies a number of auxiliary strategies not shown here.

When the conclusion of a goal sequent asserts equality, one or both of whose components is a let expression, the strategies described in Figure 7 can often simplify the let expressions to produce equivalent forms on which strategies for resolving equalities may be applied directly. An example in

L1. Sort local definitions in order of their dependency.
Since the order of definitions in a common scope has no semantic significance, a list of definitions can be order-sorted into a sequence consistent with the partial order of their dependencies. In case the list contains a clique of mutually recursive definitions, the relative order of the clique is preserved by this sorting.

L2. Rename variables bound in the pattern of a definition.
The strategy that renames with fresh variables all occurrences of variables bound in the pattern of a local definition cannot fail. This strategy prepares a non-recursive definition to be lifted into a surrounding context.

L3. Inline simple variable definitions.
A non-recursive definition whose left-hand side pattern is a simple variable can be inlined by capture-avoiding substitution of its right-hand side for every free occurrence of the variable in the scope of the **let**.

L4. Simplify definitions by structure matching.
When the left side of a definition is a structured pattern (i.e., not a simple variable) that is matched by the right hand side, the definition can be split into one or more, simpler definitions by structure matching. This may enable subsequent definition inlining.
Since the definitions in a **let** expression use lazy matching, structure matching succeeds only if the entire pattern on the left of a definition is matched by the expression on the right.

L5. Lift independent local definitions into an enclosing scope.
A definition can be lifted from a **let** expression into an enclosing scope if its right hand side contains no occurrence of a variable bound in the pattern of another definition in the same scope.

L6. Eliminate redundant let forms.
A **let** expression, all of whose definitions have been eliminated by inlining and/or lifting to a surrounding scope, is redundant and can be rewritten to its object expression alone.

Fig. 7. Strategies for simplifying expressions containing **let** definitions

which these strategies are used to determine a non-obvious equivalence of two let expressions has been given in [5].

In the following section, the example chosen to illustrate strength induction asserts the equality of a compound `let` expression to a variable.

8 Strength induction

Program verification for Haskell is inherently more complex than would be the case for a comparable language with strict semantics. This is because certain Haskell contexts, such as that of a **case** scrutinee, an operand of a strict operator or an argument of a data constructor declared with a strictness annotation, require a term in that context to be well-defined under evaluation, whereas other contexts do not. Unlike a language with call-by-value semantic throughout, one cannot assume that free variables occuring in a Haskell term must denote well-defined values in order for the term to denote a value; the degree of definedness required depends upon the contexts of their occurrences.

Consider, for example, the assertion `M_Id2` of the module displayed in Ta-

```
module State
where
newtype State s a = ST (s -> (a, s))

-- instance Monad (State s) where
return x      = ST (s -> (x, s))
(ST c)>>=f = ST (s -> let (x, s') = c s
                          ST c'  = f x
                      in c' s')
{-P : -- monad laws
assert M_Id1 =
  All f, x.
    {f} ::: $(Univ -> $(ST $Univ)) ==>
        {return x >>=f} === {f x}
assert M_Id2 =
  All m. {m>>=return} === {m}
assert M_Assoc =
  All f, g, m.
    {(m>>=f)>>=g} === {m>>=(x -> f x >>=g)}
-}
```

Haskell's *Monad* class specifies that an instance must define two functions, $return :: a \to m\,a$, and $(>>=) :: m\,a \to (a \to m\,b) \to m\,b$, where m represents the monad type constructor.

In the code above, the newtype declaration specifies a representation by functions from a type of state objects to the product of a parameter type and the state type. The data constructor, ST, has no semantic significance.

The function *return* injects its argument into the structure of the monad representation. Its definition shows that the function specified by $return\,x$ does not change the state component.

The operator (>>=) defines function application in the monad. An application $m >>= f$ produces a function that evaluates the monadic structure of m by applying it to a state variable to produce a (value, state) pair. The components of this pair are passed as arguments to f, which returns a new (value, state) pair.

Fig. 8. A simple state monad in Haskell with asserted laws

ble 8. After elaborating the definitions and reducing applications, the assertion is represented by the sequent:

$$\vdash \left\{ \begin{array}{l} \textbf{let}\ ST\,c = m \\ \textbf{in}\ ST\,(\lambda s \text{ -> } \textbf{let}\ (x, s') = c\,s \\ \qquad\qquad\qquad\qquad ST\,c' = ST\,(\lambda r \text{ -> } (x, r)) \\ \qquad\qquad\quad \textbf{in}\ c'\ s') \end{array} \right\} \text{===} \{m\}$$

An obvious strategy for simplifying the left hand side is to try structural splitting of the variable m, i.e. to assume there is a variable d such that $\{m\}$ === $\{ST\,d\}$. However, for splitting to be a sound strategy, the condition

that m is well-defined, i.e. that the proposition m ::: $Univ, must be verified. No such property has been assumed of m in the assertion `M_Id2`.

In such a circumstance, Plover will try a strategy that we call *strength induction*. This strategy is sound at all types, because in Haskell semantics every type frame is a pointed cpo. Here's the strategy. A free variable, m, can be assumed to have either of two mutually exclusive properties which cover its possible semantic valuations; either m === *undefined* or else m ::: $Univ. The strength induction strategy makes two copies of a conjectured sequent in which a variable m occurs free, appending m === *undefined* to the assumptions of one and m ::: $Univ to the assumptions of the other. The original sequent (without the added assumptions) is discharged if both of the amended copies are discharged.

Let's try strength induction on the example.

Case 1: assume m === *undefined*

$$\{m\} === \mathit{undefined}\} \vdash \left\{\begin{array}{l}\textbf{let } ST\,c = \mathit{undefined}\\ \textbf{in } ST\,(\lambda s \texttt{->} \textbf{let } (x,s') = c\,s\\ \qquad\qquad\qquad\qquad ST\,c' = ST\,(\lambda r \texttt{->} (x,r))\\ \qquad\qquad\qquad \textbf{in } c'\,s')\end{array}\right\} === \{\mathit{undefined}\}$$

The **let** expression on the left side of the asserted equality reduces immediately to *undefined*, yielding a reflexive equality.

Case 2: assume m ::: $Univ

The condition assumed for this case enables a structural splitting strategy to append the equality $\{m\} === \{ST\,d\}$ to the assumptions of the sequent, where d is a fresh variable. Since the data constructor ST is declared in a **newtype** declaration, it is strict in its argument, thus the assumption m ::: $Univ also implies the assumption d ::: $Univ.

Structural matching (L4) and inlining (L3) eliminate the first **let** definition from the left hand term of the asserted equality, leaving

$$\{m\} ::: \$\mathsf{Univ},\ \{m\} === \{ST\,d\} \vdash \left\{\begin{array}{l}ST\,(\lambda s \texttt{->} \textbf{let } (x,s') = d\,s\\ \qquad\qquad\qquad\qquad ST\,c' = ST\,(\lambda r \texttt{->} (x,r))\\ \qquad\qquad\qquad \textbf{in } c'\,s')\end{array}\right\} === \{ST\,d\}$$

Structural matching of terms on both sides of the equality predicate further simplifies the sequent to:

$$\{m\} ::: \$\mathsf{Univ},\ \{m\} === \{ST\,d\} \vdash \left\{\begin{array}{l}\lambda s \texttt{->} \textbf{let } (x,s') = d\,s\\ \qquad\qquad\qquad\qquad ST\,c' = ST\,(\lambda r \texttt{->} (x,r))\\ \qquad\qquad\quad \textbf{in } c'\,s'\end{array}\right\} === \{d\}$$

A strategy for verifying an asserted equality, one of whose terms is an application (Sec. 6.2), applies both terms to a common, fresh variable and attempts to reduce the resulting applications. The sequent is transformed to:

$$\{m\} ::: \$\mathsf{Univ}, \{m\} === \{ST\,d\} \vdash \left\{\begin{array}{l}\textbf{let } (x, s') = d\,s_1 \\ \quad\quad ST\,c' = ST\,(\lambda r \texttt{->} (x, r)) \\ \textbf{in } c'\ s'\end{array}\right\} === \{d\,s_1\}$$

At this point, the **let** expression on the left might be simplified by using strategies (L1–L6) of Figure 7. However, to apply L2, structure matching, to simplify the definition $(x, s') = d\,s_1$, the definiens must be assumed equivalent to a normal form, i.e. $d\ s_1 === (x_1, x_2)$. It is sound to introduce such an assumption only if the expression $d\,s_1$ is well-defined.

Since no property has been assumed of $d\,s_1$ other than its type, the deduction appears to have reached a dead end. However, a second application of strength induction saves the day.

Case 2.1: assume $d\,s_1 === \mathit{undefined}$. In this case, the **let** expression on the left side of the equality reduces immediately to *undefined* and the sequent is discharged by reflexive equality.

Case 2.2: assume $d\ s_1 ::: \$\mathsf{Univ}$. The assumed condition enables structure splitting. It can be assumed that $d\,s_1 === (x_1, x_2)$, where x_1 and x_2 are fresh variables. Under this assumption, strategies L3–L5 eliminate the **let** expression. The sequent is rewritten to:

$$\{m\} ::: \$\mathsf{Univ}, \{m\} === \{ST\,d\}, \{d\,s_1\} ::: \$\mathsf{Univ}, \{d\,s_1\} === \{(x_1, x_2)\} \vdash \{(\lambda r \texttt{->} (x_1, r))\ x_2\} === \{(x_1, x_2)\}$$

Reducing the application on the left side of the concluded equality results in a reflexive equality, allowing the sequent to be discharged. The Nelson-Oppen decision procedure for equality [6] has been implicitly used in elaborating the example.

9 Conclusions

We have given a brief overview of strategies used in Plover, an automatic verification tool for properties of Haskell98 programs. The power of strategies for controlling rewriting is illustrated by the example of normalization strategies to achieve three, different normal forms for a simple lambda calculus. However, Plover employs far more than normalization to verify Haskell programs.

The difficulties presented by mixed evaluation rules (strict and non-strict) are the principal topic of this paper. We have given a new and powerful strategy called strength induction, which resolves well-definedness conditions on

the evaluation of an expression when no assumptions of well-definedness have been stated. Use of this strategy enables Plover to verify stronger property assertions (requiring fewer assumptions) than had previously been possible.

Surprisingly, there has been relatively little prior work on the definedness aspect of programs in languages with mixed-evaluation semantics. The problem has been recognized by the architects of *Sparkle* [2], the verifier for *Clean 2* [1,9]. One approach to automating reasoning about definedness properties is suggested by van Eekelen and de Mol [12] who describe a *definedness tactic* that has been incorporated into *Sparkle*. Insofar as we can determine, the strength induction strategy presented in this paper appears to be similar in concept to the definedness tactic, apart from from a difference in notation.

References

[1] Tom Brus, Marko van Eekelen, Maarten van Leer, and Rinus Plasmeijer. Clean: a language for functional graph rewriting. In *Proc. of the Third Internat. Conf. on Functional Programming Languages and Computer Architecture (FPCA'87)*, volume 274 of *LNCS*, pages 364–384. Springer Verlag, 1987.

[2] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers—SPARKLE: A functional theorem prover. In *Proc. of 13th Internat. Workshop on Implementation of Functional Languages (IFL'01)*, volume 2312 of *LNCS*, pages 99–118. Springer Verlag, 2001.

[3] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[4] Richard B. Kieburtz. P-logic: Property verification for Haskell programs. ftp://ftp.cse.ogi.edu/pub/pacsoft/papers/Plogic.pdf, 2002.

[5] Richard B. Kieburtz. Programmed strategies for verification. In Sergio Antoy, editor, *Sixth International Workshop on Reduction Strategies in Rewriting and Programming*, Electronic Notes in Theoretical Computer Science. Elsevier, 2007.

[6] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer Verlag, 2002.

[8] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th Internat. Conf. on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[9] Rinus Plasmeijer and Marko van Eekelen. Clean Version 2.0 Language Report. http://clean.cs.ru.nl/download/Clean20/doc/CleanRep2.0.pdf, December 2001.

[10] The *Logical* team. The Coq proof assistant. http://coq.inria.fr,2006.

[11] M. G. J. van den Brand, H. A. de Jong, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.

[12] Marko van Eekelen and Maarten de Mol. Proof tool support for explicit strictness. In *Proc. of 17th Internat. Workshop on Implementation of Functional Languages (IFL'05)*, volume 4015 of *LNCS*, pages 37–54. Springer Verlag, 2005.

[13] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.